# Relational Databases and Web Integration

*Dr. Carl Pulley*
*c.j.pulley@hud.ac.uk*

Monday, 17 August 2009

This weeks lecture notes are based upon Part II: Building an Application of the course recommended text: Agile web Development with Rails (3rd Edition) by S.Ruby, D.Thomas and D.H.Hansson.

# Integration Testing

- Here we aim to exercise the *flow* through our application
- We can do this by writing tests that implement the use-case scenarios or stories that our client will have used to describe the application

# A User Story

- A user goes to the store index page.
- They select a product, adding it to their cart.
- They then check out, filling in their details on the checkout form.
- When they submit, an order is created in the database containing their information, along with a single line item corresponding to the product they added to their cart.

# Story Implementation

```
script/generate integration user_stories
...

class UserStoriesTest < ActionController::IntegrationTest
  fixtures :all
  # Replace this with your real tests.
  test "the truth" do
    assert true
  end
end
```

:products

# Story Implementation

```
test "buying a product" do
  # A user goes to the store index page.

  # They select a product, adding it to their cart.

  # They then check out, filling in their details on the checkout form.

  # When they submit, an order is created in the database containing their information,
  # along with a single line item corresponding to the product they added to their
  # cart.
end
```

# Story Implementation

```
test "buying a product" do
  ruby_book = products(:ruby_book)

  # A user goes to the store index page.

  get "/store/index"
  assert_response :success
  assert_template "index"

  ...
end
```

# Story Implementation

```
test "buying a product" do
  ...

  # They select a product, ...

  put "/store/add_to_cart", :id => ruby_book.id
  assert_response :success
  assert_template "add_to_cart"

  # ... adding it to their cart.

  cart = session[:cart]
  assert_equal 1, cart.items.size
  assert_equal ruby_book, cart.items[0].product

  ...
end
```

# Story Implementation

```
test "buying a product" do
  ...

  # They then check out, ...

  post "/store/checkout"
  assert_response :success
  assert_template "checkout"

  # ... filling in their details on the checkout form.

  post_via_redirect "/store/save_order", :order => { :name => "Dave Thomas",
     :address => "123 The Street", :email => "dave@pragprog.com", :pay_type => "check" }
  assert_response :success
  assert_template "index"
  assert_nil session[:cart]

  ...
end
```

# Story Implementation

```
test "buying a product" do
  ...

  # When they submit, an order is created in the database containing their information,
  # along with a single line item corresponding to the product they added to their
  # cart.

  orders = Order.find(:all)
  assert_equal 1, orders.size
  order = orders[0]

  assert_equal "Dave Thomas", order.name
  assert_equal "123 The Street", order.address
  assert_equal "dave@pragprog.com", order.email
  assert_equal "check", order.pay_type

  assert_equal 1, order.line_items.size
  line_item = order.line_items[0]
  assert_equal ruby_book, line_item.product
end
```

# Story Implementation

```
test "buying a product" do
  LineItem.delete_all
  Order.delete_all

  ruby_book = products(:ruby_book)

  ...
end
```

# Singletons

- Using singletons we can build up a domain specific testing language
- Singleton methods are defined on single objects **not** classes
- Can do this with the following Ruby syntax:

```
def obj.method_name(args)
  ...
end
```

# Our DSL

- Can identify 4 common verbs/actions:
  - viewing a page
  - buying a product
  - cart contents
  - checking out

# DSL Testing Language

```
def regular_user
  open_session do |user|
    def user.is_viewing(page)
      assert_response :success
      assert_template page
    end

    def user.buys_a(product)
      put "/store/add_to_cart", :id => product.id
      assert_response :success
      assert_template "add_to_cart"
    end

    def user.has_a_cart_containing(*products)
      cart = session[:cart]
      assert_equal products.size, cart.items.size
      for item in cart.items
        assert products.include?(item.product)
      end
    end

    ...
  end
end
```



Here, open_session opens up a new testing session object (captured by the variable user) which the method regular_user returns as its result.
The DSL based methods (ie. is_viewing, etc.) are defined on this testing session object.

# DSL Testing Language

```
def regular_user
  open_session do |user|
    ...

    def user.checks_out(details)
      post "/store/checkout"
      assert_response :success
      assert_template "checkout"
      post_via_redirect "/store/save_order", :order => { :name => details[:name],
                        :address => details[:address], :email => details[:email],
                        :pay_type => details[:pay_type] }
      assert_response :success
      assert_template "index"
      assert_nil session[:cart]
    end
  end
end
```

# DSL Tests

```
DAVES_DETAILS = { :name => "Dave Thomas", :address => "123 The Street",
                  :email => "dave@pragprog.com", :pay_type => "check" }

MIKES_DETAILS = { :name => "Mike Clark", :address => "345 The Avenue",
                  :email => "mike@pragmaticstudio.com", :pay_type => "cc" }

def setup
  LineItem.delete_all
  Order.delete_all
  @ruby_book = products(:ruby_book)
  @rails_book = products(:rails_book)
end

def test_buying_a_product
    dave = regular_user
    dave.get "/store/index"
    dave.is_viewing "index"
    dave.buys_a @ruby_book
    dave.has_a_cart_containing @ruby_book
    dave.checks_out DAVES_DETAILS
    dave.is_viewing "index"
    check_for_order DAVES_DETAILS, @ruby_book
end
```

# DSL Tests

```
def test_two_people_buying
  dave = regular_user
  mike = regular_user
  dave.buys_a @ruby_book
  mike.buys_a @rails_book
  dave.has_a_cart_containing @ruby_book
  dave.checks_out DAVES_DETAILS
  mike.has_a_cart_containing @rails_book
  check_for_order DAVES_DETAILS, @ruby_book
  mike.checks_out MIKES_DETAILS
  check_for_order MIKES_DETAILS, @rails_book
end
```

# Behavior Driven Development

- BDD is a development regime where by one uses a DSL to specify the expected behavior that the intended application is to have
- One uses use-case scenarios to drive the implementation of integration tests using a DSL
- RSpec and Cucumber are two BDD frameworks that support this style of code development